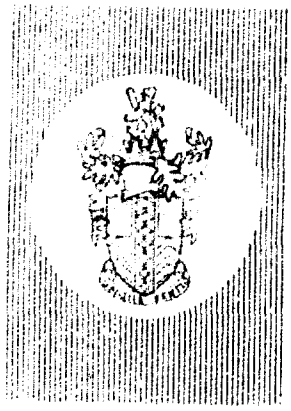
UNLIMITED

BR110656

(2)

Report No. 88014



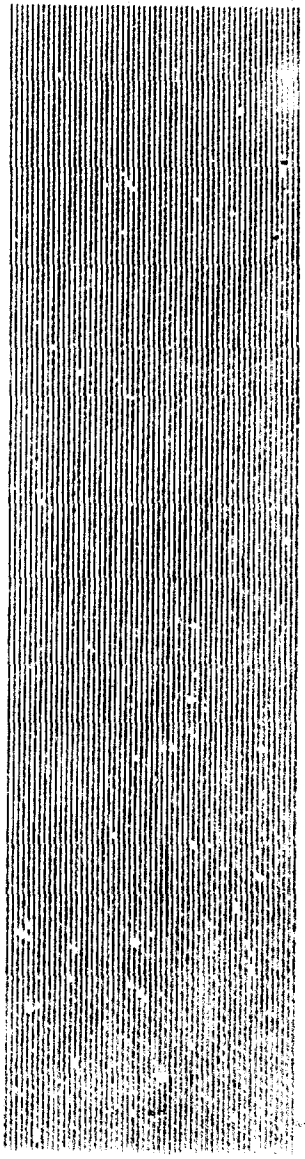
Report No. 88014

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

AD-A210 022

KEEPSAKE: A DATABASE KERNEL

Authors: N E Pealing & K R Milner



PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

March 1989

UNLIMITED

CONDITIONS OF RELEASE

0040071 BR-110080

********** U

********** Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

Royal Signals and Radar Establishment

Report 88014

Title: KeepSake : A Database Kernel

Authors: N.E. Peeling, K.R. Milner

Date: December 1988

Summary

KeepSake is a set of disc management procedures that can be used to extend a programming language to provide data persistence. It provides efficient low-level read/write procedures and also allows flexibility in partitioning a database for garbage collection and multi-user read/write access. KeepSake does not impose a data schema on the user but can be used to support a number of database types (relational, hierarchical etc.).

THIS PAGE IS LEFT BLANK INTENTIONALLY

Contents

THIS PAGE IS LEFT BLANK INTENTIONALLY

KeepSake : A Database Kernel

Introduction

KeepSake is a set of disc management procedures that can be used to extend a programming language to provide data persistence. It provides efficient low-level read/write procedures and also allows flexibility in partitioning a database for garbage collection and multi-user read/write access. KeepSake does not impose a data schema on the user but can be used to implement a number of database types (relational, hierarchical etc.).

KeepSake has been developed over a number of years at the Royal Signals and Radar Establishment and is a direct successor of a single-user system called ADAM[1], which was written to provide a secure basis for the support environment of the Hardware Description Language ELLA[2]. KeepSake is implemented as a non-overwriting system, which greatly simplifies the problem of maintaining data integrity and also allows one person to update a section of the database at the same time that it is being read by one or more users.

1 Non-overwriting Systems

Most database systems are written in an overwriting way - in other words altering the data in such a database means that existing data is overwritten on disc and all database transactions are logged in a separate file so that a consistent database state can be recovered in the case of a machine failure. This is a safe way of implementing a database, but the software needed to guarantee data integrity will be complex. For example, if there is a machine crash, the database attempts to recover a consistent state by rewriting the original data and there is another machine failure during this process, then the database will be left in an inconsistent state unless the recovery process is also logged. Hence to guarantee integrity, a set of log files is needed.

Our approach tackles the problem of data recovery after a machine failure in quite a different way: whenever a new block of data is written, KeepSake finds a new piece of disc - it does not overwrite existing data. The advantage of this method is that since the old data is not overwritten, data integrity can be maintained without resorting to log files.

A KeepSake data structure on disc consists of a network of data and pointers. Whenever a new block of data is to be written, KeepSake finds a new piece of disc, writes away the data and delivers a pointer to the data (pointers in KeepSake are known as DISCPTRs). In this way, the user creates a structure accesible by a single DISCPTR known as the root (see Figure 1). He then calls a KeepSake 'finish' procedure that writes away his root.

Now consider the problem of updating a KeepSake database, for example by replacing "abc" by "def" in Figure 1, leaving the rest of the database unaltered. Firstly, the user writes away his new data and receives a new DISCPTR to his data. He then constructs his new database by propagating the change back to the new root, using the new DISCPTR and DISCPTRs from the old database. The new root is written away with a call of "finish" as before. The new database is illustrated in Figure 2.

If there is a machine failure before the new root has been written away, then the user can revert to the old state of the database, since his old data has not been overwritten.

Figure 1: A KeepSake Database Structure

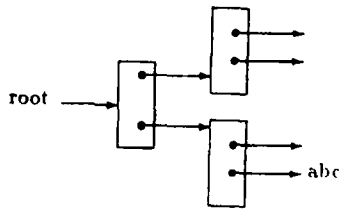
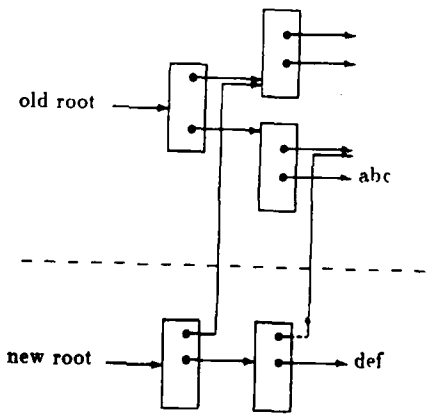


Figure 2: Altering a KeepSake Database



The only dangerous situation is if there is a machine crash when writing away the new root - we cope with this possibility by writing away the new root three times. This allows KeepSake to detect a failure when the file is next opened and to return a value that the user can check for, which tells him whether or not his last update succeeded. If not, the old root can safely be used, since the old data has not been overwritten.

Data that is not accessed by the new root can only be recovered by garbage collection - there is no explicit delete operation. This means that it is impossible to delete data from one part of the database that is still accessed elsewhere. We also provide another way of updating a KeepSake database (using a type of pointer known as a DISCPTR variable) which allows a change to be made without having to propagate changes back to the root. This is explained in the next section.

One of the main criticisms of non-overwriting systems has been that they are inefficient in terms of disc space used. Experience with the Flex system[3] developed at RSRE which also uses a non-overwriting approach has shown this not to be the case. For example, take the process of editing a text file. On most conventional overwriting systems, when a text file is edited, a copy of the file is made and the user edits the copy, keeping the original version intact in case of a machine failure. If the file is large and the changes few, this can be very wasteful in disc space. This is particularly true if the user requires more than one generation of his file to be preserved - multiple copies of the entire file will be stored, when it is only the minor differences between versions that are important.

A better alternative is to create a log file to record all changes made to the text file. This means that data integrity can be maintained without keeping a copy of the text file, but the disadvantage of this approach is that three disc accesses are needed for every change made to the text file - one to read the *old* data, one to write the old data to the log file and one to write the new data to the text file.

Our non-overwriting approach requires neither a file copy nor a log file to monitor changes to the text. If the text file is mapped onto a KeepSake disc structure of a hierarchy of sections and subsections with the text stored at the leaf nodes of the hierarchy, small changes in the text file will produce few changes in the underlying disc structure, since any data that is unchanged between versions will not be rewritten. This also allows previous generations to be stored much more efficiently - instead of complete copies of different version being stored, effectively only the differences between versions are stored.

2 DISCPTRs

KeepSake provides three basic types of DISCPTR: DISCPTR values, DISCPTR variables and shaky DISCPTRs. All three types of DISCPTR can be incorporated into a structure on disc and read using the same KeepSake read procedure; they have slightly different properties which affect garbage collection and/or the way in which a KeepSake database can be updated.

DISCPTR values are constant references; they always point to the same data and cannot be altered. A database consisting only of DISCPTR values can only be altered by creating a new DISCPTR, incorporating it into the database and propagating all changes back to the root (as described in the previous section). The change will then only take effect when the new root has been written. Note that only hierarchical (acyclic) data structures can be built using DISCPTR values

DISCPTR variables, as the name suggests, can be assigned to. The property of assignment means that cyclic structures (networks) can be built on disc. Variables also allow a database to be updated without having to propagate the change back to the root - the new data can simply be assigned to an existing DISCPTR variable. The assignment is written to disc when a new database state is created by a call of the KeepSake 'finish' or 'commit' procedures. 'Finish' updates the database, writing away all DISCPTR variables and a new root, 'commit' just writes away the new variables (it is effectively a finish called on the old root). Note that updating DISCPTR variables does not involve overwriting on disc - implementation details are given in [4].

Another property of DISCPTR variables is that any copy of a variable will automatically access any new data assigned to the original variable. With DISCPTR values this is not true - to achieve the same effect, the user must search for all copies of his original DISCPTR, substitute the new DISCPTR for each copy of the original and propagate all these changes back to the root - a rather long-winded process!

Shaky DISCPTRs can only be created by a call of 'make_shaky', which takes an existing DISCPTR value and produces a new shaky DISCPTR (shaky DISCPTR variables are not allowed). The difference between a shaky and non-shaky DISCPTR is defined by garbage collection: a shaky DISCPTR will only be kept alive if a non-shaky DISCPTR to the same data is still alive; shaky DISCPTRs have no independent life of their own. If an attempt is made to use a shaky DISCPTR when the original pointer has been garbage collected and the space freed, then the shaky will return a special null value (which can be tested for), otherwise shaky DISCPTRs can be used in exactly the same way as non-shakies.

A shaky DISCPTR can be used when one user wishes to provide others with access to a block of data yet still maintain sole control over when that data is disposed of by garbage collection. By keeping a non-shaky DISCPTR himself and providing other users with shaky copies, he can ensure that the data will be collected when his non-shaky is no longer accessible, regardless of the accessibility of any shaky copies. The shaky copies will continue to access the data until it is disposed of by garbage collection and afterwards will access the special null value described above.

All non-shaky copies of a DISCPTR have the same status as the original - in other words a block of data will only be collected if *all* non-shaky DISCPTRs to it are no longer accessible.

Note that there is an instore overhead of 22 bytes for each DISCPTR variable and shaky DISCPTR in a region open for reading.

3 Data Manipulation

Writing and Reading Data

There is a single KeepSake data primitive which has two components: an array of DISCPTRs and an array of characters (CHARs). This enables all the read/write procedures that were available in ADAM to be produced very easily in KeepSake ; also the separation of CHARs from DISCPTRs is useful for garbage collection, since our garbage collection algorithm only requires us to read DISCPTR blocks from disc - this is explained more fully in [4].

We have provided two levels of read/write procedure - level one contains just a single read and single write procedure. The write procedure takes a vector of CHARs and a vector of DISCPTRs (a vector is a contiguous array indexed from 1) and delivers a DISCPTR. This DISCPTR can be either a DISCPTR value or variable and can itself be written to disc. The read procedure does the reverse: it takes a DISCPTR and assigns the CHARs and DISCPTRs found into (user-provided) arrays of CHARs and DISCPTRs.

The level two interface is a straightforward implementation of the current ADAM read/write procedures, on top of level one. This enables a current user of ADAM to convert to KeepSake with a minimum of effort.

Appending Data

Data can also be written away by an 'append' operation. This consists of a single call of 'start_append', which writes away a VECTOR [] CHAR and delivers a DISCPTR, a number of calls of 'append', which extend this VECTOR [] CHAR and a call of 'finish_append', which completes the append by writing a VECTOR [] DISCPTR. The effect is the same as if the user had written away the extended character array and the VECTOR [] DISCPTR as a single call of the level one write procedure, but using append in this way means that the user can write away his data as it is generated. Note that no other writes are allowed during an append - any that are attempted will fail.

Overwriteable Sectors

It is sometimes more convenient to use the more conventional overwriting strategy; a good example is implementing a 'redo' log, in which the user takes the old state of the database after a machine failure and reconstructs a sequence of transactions from a log file up to the time of failure. In this example, we require a record of all transactions up to the time of failure and the obvious way to implement this is with an overwriteable file; KeepSake effectively allows the user to embed such a file into his database via an overwriteable sector.

An overwriteable sector of disc is quite separate from the rest of the KeepSake database; any data written to it cannot affect the integrity of the rest of the database. It provides the user with 'raw' disc, which he has direct access to and is incorporated into the KeepSake database via a DISCPTR like any other block of data, allowing it to be garbage collected. The only difference is that the integrity of the overwriteable sector of disc is not guaranteed in the case of a machine crash, since the old data may have been partially overwritten.

4 KeepSake Database Structure

A KeepSake database can be partitioned into regions, each of which allows simultaneous single write and multiple read access. Each region consists of a network of data and KeepSake pointers (referred to as DISCPTRs), accessible by a single root DISCPTR. The regions in a KeepSake database are arranged in a hierarchy: each region can contain a number of sub-regions, but will only have one 'parent' region. The uppermost region in the hierarchy is created automatically (with the same name as the database) when the database is created.

Although each region has only a single root, data written in one region can be incorporated into another via a DISCPTR (provided that the scope rules are not violated). Hence a database structure can be accessed by another region at any node.

KeepSake allows two or more regions to be updated together as a single atomic operation. In this way KeepSake allows the user to link two or more regions together - either all the regions involved in the update will reach their new states, or they will all revert to their old states.

Data Integrity

There are two aspects to the internal consistency of a KeepSake database; firstly, the problem of recovering from a session that was aborted (e.g. due to a machine crash) and secondly, maintaining data consistency across regions.

Recovery from an aborted session is automatic. The next time an open is attempted on a region involved in a failed update, KeepSake will check for internal inconsistencies and revert to the previous state of the region if necessary, indicating this to the user via a status flag. Note that since KeepSake is a non-overwriting system, it has no need to keep a record of all transactions in a session to be able to revert to the old state of the database.

Maintaining data consistency across regions is a separate problem. KeepSake allows a number of regions to be updated in a single atomic operation and guarantees that such atomic operations cannot subsequently be undone. To illustrate the problem, consider two users accessing regions A and B: the first user opens A for reading and the second user who has write access to A and B then atomically updates A and B to produce A' and B', whereupon the first user attempts to open B' for reading. If this open were allowed to succeed, then the first user would have access to (A, B') and would therefore have undone the atomic update that produced (A', B') from (A, B). Any attempt to open a region in such a way as to undo an atomic operation will cause a failure which will identify the inconsistency together with the operation(s) that will produce a consistent set of regions (in the example given, consistency can be restored by re-opening region A).

5 Hierarchy and Scopes

The hierarchy of regions forms a *partial order* on a KeepSake database. If R and S are both regions, then:

$R < S \Rightarrow$ R is a descendant of S

Note that this is only a partial order; 'sibling' regions (regions with the same parent region) are not considered to be equal under this ordering.

DISCPTR Scopes

KeepSake uses the hierarchy of regions to control DISCPTR scopes. The scope of a DISCPTR is defined as the set of regions into which it may be written; the scope is defined when the DISCPTR is created and cannot subsequently be changed. Whenever a DISCPTR is created in a region R, a 'scope' region is specified, which must be either R or an ancestor of R.

If R is a region, then define:

descendants(R) = { regions S with $S \leq R$ }

If a DISCPTR dp is created in a region R with scope region S, then

scope(dp) = descendants(S) (provided that $R \leq S$)

In other words, dp can be written into the region S or any of its descendants. This is illustrated in Figure 3.

If the scope region is a region other than R (i.e. dp can be written into a region further up the hierarchy), then dp is known as an exported DISCPTR. Note that exported DISCPTRs have the same instore overhead of 22 bytes as DISCPTR variables and shaky DISCPTRs.

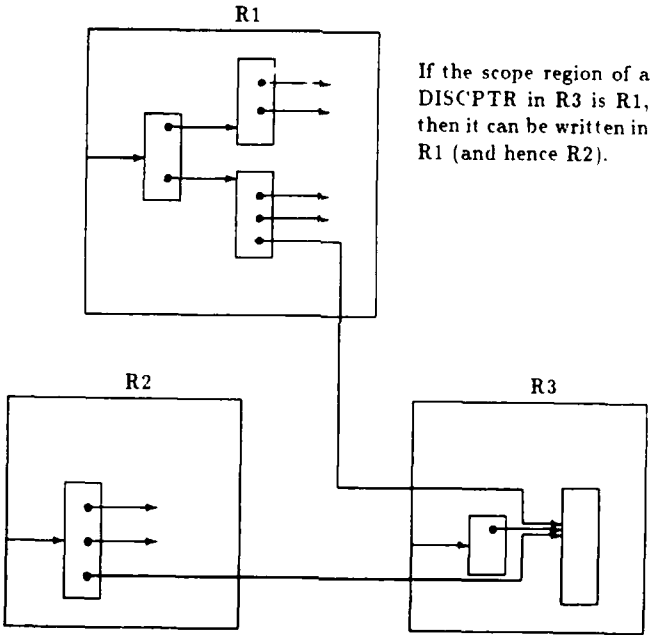
Region Scopes

The restrictions on DISCPTR scopes described in the previous section can be enforced by applying a similar sort of restriction to a region when creating it. A region scope imposes an upper bound on the scope of all DISCPTRs created within the region, and any attempt to create a DISCPTR with a greater scope than this upper bound will fail.

regionscope(R) = SC =

$\forall$ DISCPTRS dp created in region R
scope(dp) $\leq$ descendants(SC)

Figure 3: DISCPTR scopes



If the scope region of a DISCPTR in R3 is R1, then it can be written in R1 (and hence R2).

6 Garbage Collection

KeepSake allows considerable flexibility in garbage collection; the user can collect his entire database, or a subset of regions, or in some cases just a single region. We provide two garbage collectors: compacting and non-compacting. Both free disc space that is no longer part of the database structure: the non-compactor only needs to read pointer blocks from disc - we anticipate it to be fast enough to form part of an interactive system. Over a period of time however, if only the non-compactor is called, there will be a gradual degradation of performance over time, since the freelist will become more and more fragmented. Also, since the disc is divided into units of 512 bytes and we allow more than one data-block per unit, it is possible for a unit of disc to be only partially used (a unit of disc can only be freed for re-use by garbage collection if all the datablocks it contains are freed). This means that the total amount of free disc space will tend to decrease with time.

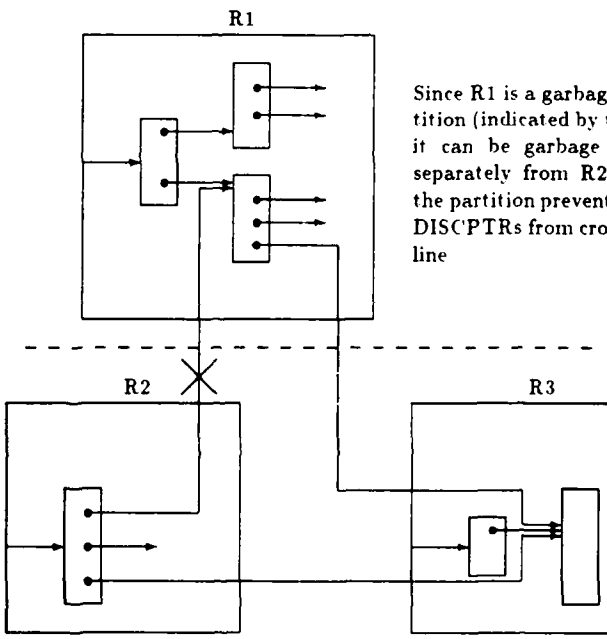
These problems are addressed by the compactor: it reads the data structure in each region being collected and rewrites it into a new file, removing any discontinuities in the freelist. Compaction will be considerably slower then non-compaction since it has to read in and rewrite entire regions and so we would anticipate it being run as an offline procedure.

Garbage Collection Partitions

KeepSake also provides a way of restricting DISCPTR scopes which allows the garbage collector to collect the top half of a tree of regions separately from the bottom half. Whenever a region R is created, it is possible to make it a garbage collection partition. This means that KeepSake will prevent DISCPTRs in R (or any of its ancestor regions) from being written into a descendant of R.

Garbage collection partitions provide the user with an additional option for garbage collection. When called on a region R, the user can either collect R and all its descendants, or R and all regions in each branch of the hierarchy below R down to (and including) the first region which is a garbage collection partition. This second alternative is quite safe since the garbage collector knows when it collects a garbage collection partition that no DISCPTRs from further down the hierarchy of regions need to be traced to protect data in that region or any of its ancestors. This is illustrated in Figure 4.

Figure 4: Partitioning Garbage Collection



Since R1 is a garbage collection partition (indicated by the dotted line), it can be garbage collected quite separately from R2 and R3, since the partition prevents any 'upwards' DISC'PTRs from crossing the dotted line

7 The Procedural Interface

KeepSake is currently available in ALGOL68, but we anticipate no great problems in translating it to other high-level languages as and when appropriate. The following sections describe the current ALGOL68 interface.

7.1 Create and Open PROCs

A KeepSake database is created by create_db and subsequently opened by open_db:

```
PROC create_db = (VECTOR [] CHAR db_name) DATABASE:

PROC open_db = (VECTOR [] CHAR db_name) DATABASE:

PROC create_region = (REF DATABASE db, ACCESS parent, max_scope,
                      BOOL gc_partition, VECTOR [] CHAR name) VOID:
```

where "parent" is the region in which the region is to be created and "max_scope" is the region which limits the scope of any DISCPTR produced by the new region. Regions can be specified either by giving the name of the region or a DISCPTR from the region (ACCESS = UNION(VECTOR [] CHAR, DISCPTR)). "Gc_partition" indicates whether or not the region is to be a garbage collection partition or not. Each call of create_region creates three associated files - one is the file in which data is written, the other two (which contain no data) control read and write access to the data file.

Get_root delivers the root pointer from a region:

```
PROC get_root = (REF DATABASE db, VECTOR [] CHAR name) DISCPTR:
```

A number of regions can be opened simultaneously by open:

```
PROC open = (REF DATABASE db, VECTOR [] OPENACCESS regions)
                                        VECTOR [] STATUS:
```

(where OPENACCESS (= STRUCT (ACCESS region, BOOL writeable)). Regions can be opened by specifying either the name of the region or by giving a DISCPTR from the region. Open delivers an ALGOL68 structure for each region opened (STATUS = UNION (VECTOR [] VECTOR [] CHAR, INT)) - either an INT, where

0 = no errors detected
1 = last update of this region failed - previous version is being used
2 = region being written to by another user

or a VECTOR [] VECTOR [] CHAR which gives the names of the regions that must be re-opened to avoid database inconsistency (this is explained in Section 4).

7.2 Read/Write PROCs

We provide two levels of read/write procedure - level two is written in terms of level one and is provided for compatibility with ADAM, the current single-user kernel. Levels one

and two can be used in the same program, but a level one procedure cannot be used to read data written by a level two procedure (and vice versa).

Level One PROCs

The basic level one write procedure is write_chars_dps:

```
PROC write_chars_dps = (REF DATABASE db, VECTOR [] CHAR chars,
                        VECTOR [] DISCPTR dps, ACCESS region, scope,
                        BOOL assignable) DISCPTR:
```

where "region" is the region the data is to be written to, "scope" gives the scope of the DISCPTR produced and "assignable" indicates whether the DISCPTR is a value or a variable.

The user can check whether there is enough space on disc to write away his data using can_write:

```
PROC can_write = (REF DATABASE db, VECTOR [] CHAR chars,
                  VECTOR [] DISCPTR dps, ACCESS region) BOOL:
```

Characters can be appended onto an existing datablock with the three append PROC's: the block is created by start_append, can be appended to any number of times and must be completed by a call of finish_append. Note that no other disc writes can be performed during an append operation.

```
PROC start_append = (REF DATABASE db, VECTOR [] CHAR data,
                     ACCESS region, scope, BOOL assignable) DISCPTR:

PROC append = (REF DATABASE db, VECTOR [] CHAR data, DISCPTR onto)
                                                        DISCPTR:

PROC finish_append = (REF DATABASE db, VECTOR [] CHAR data,
                      VECTOR [] DISCPTR dps, DISCPTR onto) DISCPTR:
```

The level one read procedure takes a DISCPTR and fills both arrays with the data pointed to.

```
PROC read_chars_dps = (DATABASE db, REF VECTOR [] CHAR chars,
                       REF VECTOR [] DISCPTR dps, DISCPTR dp) VOID:
```

The size of the arrays "chars" and "dps" assigned to by read_chars_dps are provided by num_chars and num_discptrs respectively.

```
PROC num_chars = (REF DATABASE db, DISCPTR dp) INT:

PROC num_discptrs = (REF DATABASE db, DISCPTR dp) INT:
```

Level Two PROCs

There are three level two write PROCs which write away VECTORs of INTs, CHARs and DISCPTRs:

```
PROC write_ints = (REF DATABASE db, VECTOR [] INT data,
                                ACCESS region, scope, BOOL assignable) DISCPTR:

PROC write_chars = (REF DATABASE db, VECTOR [] CHAR data,
                                ACCESS region, scope, BOOL assignable) DISCPTR:

PROC write_discptrs = (REF DATABASE db, VECTOR [] DISCPTR data,
                                ACCESS region, scope, BOOL assignable) DISCPTR:
```

The size of array to be read back from a DISCPTR created by a level two PROC is given by PROC size:

```
PROC size = (REF DATABASE db, DISCPTR dp) INT:
```

There are three level two read PROCS corresponding to the three write PROCs given earlier. Note that KeepSake will flag an error if, for example, read_chars is used to read data written by write_ints. We provide a procedure discptr_data (see Section 7.4) to interrogate DISCPTRs, which can be used before reading, to trap any potential errors of this sort.

```
PROC read_ints = (REF DATABASE db, REF VECTOR [] INT data,
                                                DISCPTR dp) VOID:

PROC read_chars = (REF DATABASE db, REF VECTOR [] CHAR data,
                                                DISCPTR dp) VOID:

PROC read_discptrs = (REF DATABASE db, REF VECTOR [] DISCPTR data,
                                                DISCPTR dp) VOID:
```

PROCs for Handling Overwriteable Sectors

Make_overwriteable produces an overwriteable sector of disc. Its size is specified by "blocks" in terms of the unit of disc provided by the host operating system (512 bytes under VMS). The DISCPTR delivered can be a variable or value depending on whether "assignable" is TRUE or FALSE:

```
PROC make_overwriteable = (REF DATABASE db, ACCESS regions, scope,
                                INT blocks, BOOL assignable) DISCPTR:
```

The data pointed to by a DISCPTR which has been produced by make_overwriteable can be overwritten; the overwriteable sector is indexed by "block". If a block of data is given which would exceed the overwriteable range, KeepSake will flag an error. Overwrite is a synchronous write to disc which is written to disc immediately (unlike other writes which are buffered).

```
PROC overwrite = (REF DATABASE db, INT block, VECTOR [] CHAR data,
                                                    DISCPTR dp) VOID:
```

To read from an overwriteable sector (indexed by "block") use read_overwriteable. If the VECTOR [] CHAR given exceeds the overwriteable sector of disc, the procedure will fault.

```
PROC read_overwriteable = (REF DATABASE db, INT block,
                           REF VECTOR [] CHAR data, DISCPTR dp) VOID:
```

7.3 Close and Finish PROCs

A number of regions in one database can be updated with new roots (and their corresponding files closed) with finish_and_close:

```
PROC finish_and_close = (REF DATABASE db, VECTOR [] REGIONDP
                                                    regions) VOID:
```

where (REGIONDP = STRUCT (STR name, DISCPTR root)). Finish does likewise, without closing the files:

```
PROC finish = (REF DATABASE db, VECTOR [] REGIONDP regions) VOID:
```

To check that there is enough space on disc to call finish (or finish_and_close):

```
PROC can_finish = (REF DATABASE db, ACCESS region) BOOL:
```

To exit the database without writing away any changes made (and closing all appropriate files):

```
PROC quit = (REF DATABASE db) VOID:
```

Note that a call of finish_and_close is equivalent to a call of finish followed by a call of quit.

We also allow the user to write away all DISCPTR variables separately using commit (this is effectively a call of finish on the existing root with the new DISCPTR variables)

```
PROC commit = (REF DATABASE db, VECTOR [] ACCESS regions) VOID:
```

To check that there is enough space on disc to call commit:

```
PROC can_commit = (REF DATABASE db, ACCESS region) BOOL:
```

7.4 Miscellaneous PROCs

DISCPTR variables can be assigned to using assign_to_var (if "new" is a DISCPTR variable as well as "old", then there is an implicit coercion - the contents of "new" are assigned to "old").

```
PROC assign_to_var = (DATABASE db, DISCTPR old, new) VOID:
```

Both garbage collectors can be called on any region and collect the entire tree of regions beneath the region (all_regions=TRUE) or all regions down to (and including) garbage collection partitions.

The compactor is an off-line procedure - in other words all the relevant files must be closed otherwise a failure will occur; the non-compactor is an on-line procedure (protecting instore pointers) and all the relevant files must be already open.

```
PROC compacting_collect = (DATABASE db, ACCESS region, BOOL all_regions)
                                                                    VOID:

PROC noncompacting_collect = (DATABASE db, ACCESS region,
                                                  BOOL all_regions) VOID:
```

Nil DISCPTR values and variables can be created using make_nilptr and make_nilvar respectively:

```
PROC make_nilptr = (REF DATABASE db, ACCESS region) DISCPTR:

PROC make_nilvar = (REF DATABASE db, ACCESS region) DISCPTR:
```

Var_to_val produces a DISCPTR value from a DISCPTR variable (KeepSake flags an error if "dp" is not a DISCPTR variable):

```
PROC var_to_val = (DATABASE db, DISCPTR dp) DISCPTR:
```

Information about a discptr can be checked using discptr_type and discptr_data:

```
PROC discptr_type = (REF DATABASE db, DISCPTR dp) INT:
```

0 = not a valid discptr
1 = garbage collected shaky
2 = assignable but not shaky
3 = shaky but not assignable
4 = neither assignable nor shaky

Discptr_data indicates the type of data pointed to:

```
PROC discptr_data = (REF DATABASE db, DISCPTR dp) INT:
```

0 = not a valid discptr
1 = garbage collected shaky
2 = nilptr
3 = discptr to chars and discptrs
4 = discptr to overwriteable blocks
5 = discptr to integers
6 = discptr to chars
7 = discptr to discptrs

Where_is indicates the region in which the data has been written (not necessarily the

same as that containing the discptr pointing to it). Each region name is a pathname with a stem constructed from the names of its descendants e.g. region r3 created inside r2 inside r1 would be r1.r2.r3. Where_is splits up the name into its components and delivers the name as a vector e.g. the above example would be delivered as the vector (r1, r2, r3).

```
PROC where_is = (REF DATABASE db, DISCPTR dp) VECTOR [] RVC:
```

Shaky DISCPTRs are produced from non-shakies by a call of make_shaky:

```
PROC make_shaky = (REF DATABASE db, ACCESS region, DISCPTR dp)
                                                    DISCPTR:
```

The size of a region in blocks (units of 512 bytes) is given by region_size:

```
PROC region_size = (REF DATABASE db, ACCESS region) INT:
```

Similarly, the free disc space in a region is given by size_of_free_space:

```
PROC size_of_free_space = (REF DATABASE db, ACCESS region) INT:
```

The size of a region can be increased by a number of blocks (given by the parameter "by") using increase. Any machine failure after a call of increase but before commit or finish is dealt with when the file is next opened - KeepSake will safely use the increased file size.

```
PROC increase = (REF DATABASE db, ACCESS region, INT by) INT:
```

Same_data indicates whether or not two DISCPTRs point to the same data:

```
PROC same_data = (REF DATABASE db, DISCPTR a, b) BOOL:
```

Same_discptr indicates whether or not two DISCPTRs are the same. Note that this is not necessarily the same as the procedure same_data since two distinct DISCPTR variables may access the same data by separate assignments of the same DISCPTR.

```
PROC same_discptr = (REF DATABASE db, DISCPTR a, b) BOOL:
```

Hash_value delivers an integer value between 0 and 10000 from a DISCPTR (nilptrs produce 0). The value produced is not unique - in other words different DISCPTRs may produce the same value.

The purpose of a hash value is to allow the user to keep an instore array of DISCPTRs, indexed by some function of hash_val. The hash value is calculated from the disc address of the DISCPTR.

```
PROC hash_val = (REF DATABASE db, DISCPTR dp) INT:
```

8 A Worked Example

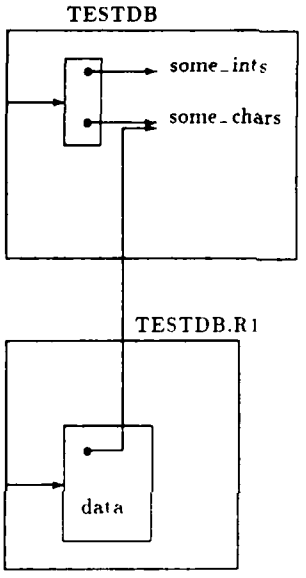
We now give an example of KeepSake in action, using a simple database with two regions. The example is split into three sessions, the first showing how to construct a

KeepSake database, the second demonstrating the read procedures and the third altering the database. The examples are fragments of ALGOL68, but should be readily understood by anyone familiar with a high-level language.

Creating a KeepSake Database

```
VECTOR [] CHAR dbname = "testdb";

    CO  Define the new database name. Note that KeepSake is not case sensitive.
    CO
  DATABASE db := create_db(dbname);


    CO  Create_db db initialises a KeepSake database, creates an outermost region
        with the same name as the database and creates all relevant files.
    CO

BOOL partition = FALSE;
create_region(db, "R1", dbname, dbname, partition);

    CO  R1 is created inside the outermost region, which is also its maximum scope
        region. "partition = FALSE" means that R1 is not a garbage collection
        partition i.e. DISCPTRs from further down the hierarchy of regions can be
        incorporated into R1's database.
    CO

VECTOR [] CHAR r1 = "TESTDB.R1";
BOOL assignable = TRUE;

VECTOR [] INT some_ints = ...
DISCPTR dp1 = write_ints(db, some_ints, dbname, dbname, NOT assignable);
VECTOR [] CHAR some_chars = ...
DISCPTR dp2 = write_chars(db, some_chars, dbname, dbname, assignable);

VECTOR [] DISCPTR dps = (dp1, dp2);
DISCPTR root_dp = write_discptrs(db, dps, dbname, dbname, NOT assignable);

    CO  A simple tree structure is built inside the outermost region "dbname" (all
        DISCPTRs created have scope given by dbname). Two DISCPTR values
        are created "dp1" and "root_dp" and one DISCPTR variable.
    CO

VECTOR [] CHAR data = ...
DISCPTR r1_root = write_chars_dps(db, data, dp2, r1, r1, NOT assignable)

    CO  The DISCPTR "dp2" from the outermost region is incorporated into R1's
        database and written away using the level one write procedure.
    CO
```

Figure 5: A Worked Example




```
VECTOR [] REGIONDP regions = ((dbname, root_dp),
                              (r1, r1_root));
finish_and_close(db, regions)

    CO  Both regions are updated with their new roots (root_dp and r1_root) in a
        single atomic operation to form the new database (see Figure5).
    CO
```

Reading a KeepSake database

```
VECTOR [] CHAR dbname = "TESTDB";
DATABASE db := open_db(dbname);

    CO  Open the database previously created.
    CO

BOOL readonly = FALSE;
VECTOR [] OPENACCESS mode = (dbname, readonly);
VECTOR [] STATUS status = open(db, mode);
```

```
CASE status[1] OF
(INT n) IF n /= 0 THEN keepsake_fault("fail on open") FI
ELSE keepsake_fault("files need re-opening")
ESAC;

    CO  The outermost region is opened for reading and the status flag tested to
        ensure that the last update completed successfully.
    CO

DISCPTR root = get_root(db, dbname);
VECTOR [size(db, root)] DISCPTR dps;
read_discptrs(db, dps, root);
VECTOR [size(db, dps[1])] INT some_ints;
read_ints(db, some_ints, dps[1]);
VECTOR [size(db, dps[2])] CHAR some_chars;
read_chars(db, some_chars, dps[2])

    CO  The data written in the outermost region is read back via the root. Note
        that the size of array needed is given by PROC size. If an attempt is made
        to read (say) INTs from a DISCPTR which points to CHARs, then this will
        be detected by KeepSake .
    CO
```

Altering a KeepSake database

```
VECTOR [] CHAR dbname = "TESTDB";
DATABASE db := open_db(dbname);
BOOL writeable = TRUE;
VECTOR [] OPENACCESS mode = (dbname, writeable);
VECTOR [] STATUS status = open(db, mode);
CASE status[1] OF
(INT n) IF n /= 0 THEN keepsake_fault("fail on open") FI
ELSE keepsake_fault("files need re-opening")
ESAC;

    CO  The outermost region is opened for writing and the status flag tested as
        before.
    CO

DISCPTR root = get_root(db, dbname);
VECTOR [size(db, root)] DISCPTR dps;
read_discptrs(db, dps, root);
VECTOR [] INT new_data = ...
BOOL assignable = TRUE;
DISCPTR new_dp = write_ints(db, new_data, dbname, dbname, NOT assignable);
assign_to_var(db, dps[2], new_dp);
```

```
CO  The contents of "new_dp" are assigned to "dps[2]". Note that the latter
    was created as a DISCPTR variable. "dps[2]" now points to INTs whereas
    previously it pointed to CHARs - KeepSake is quite happy about this. It
    is also possible to assign the contents of a DISCPTR from another region,
    provided this satisfies the restrictions on DISCPTR scopes.
CO

commit(db, dbname)

CO  All assignments are written to disc by commit. Note that the change to
    the database does not need to be propagated back to the root, since it was
    changed by an assignment to an existing DISCPTR variable.
CO
```

9 Performance Considerations

We anticipate a considerable performance improvement in KeepSake compared with its predecessor ADAM. KeepSake is much more economical in its use of disc - ADAM only wrote one data block per VAX block (512 bytes), so if the user has a large number of small data blocks a significant fraction of the disc will be wasted. KeepSake allows a number of data blocks in a single VAX block (each data block can start anywhere within a VAX block) and so there could be a significant saving on disc space with KeepSake compared with ADAM.

KeepSake has been designed to take advantage of any multi-block disc access facilites provided by the host operating system (ADAM only accessed disc by reading or writing a single VAX block). Whenever KeepSake writes away a block of data, it includes information on disc that is used by the KeepSake read procedure which works out the optimum number of VAX blocks to read in.

Our method of writing to disc in a more compact form creates a new problem: since we allow more than one data block per VAX block, a VAX block may be only partially used (since it can only be freed for re-use by the non-compactor if all the data blocks within it are no longer accessed by the database). Also, the gradual fragmentation of the freelist over time means that the multi-block reads will access fewer blocks of useful data.

Both problems are addressed by the compactor: it reads the data structure in each region being collected and rewrites it into a new file, removing any discontinuities in the freelist and rewriting the data and pointers continuously. Compaction will be considerably slower than non-compacting garbage collection; we would expect a typical garbage collection strategy to consist mainly of non-compactions with occasional offline compactions.

10 Future Work

At present a KeepSake database can only reside on one machine - our next step is clearly to produce a database kernel which allows a database to be distributed over a number of machines. Distributed databases are becoming more and more important given the rise in popularity of workstations and the relative decline of mainframes and we think our non-overwriting approach is well suited to a distributed sytem. We do not have to log all

database transactions and this makes maintaining internal consistency over a number of machines much more straightforward than with a conventional overwriting system.
We have an algorithm for performing atomic updates over several machines which is a fairly natural extension of our current single machine algorithm, but we have no plans (unfortunately!) to produce a distributed KeepSake in the near future.
KeepSake can be used to provide a persistent heap for a programming language - ADAM (our single-user version predecessor of KeepSake) has been used to produce a persistent heap for ALGOL68[5]. This could easily be adapted to use KeepSake instead of ADAM and with some alteration could be used to provide a persistent heap for other high-level languages.

11 Conclusions

We believe that KeepSake is a useful tool with which to build non-distributed databases. Most database products on the market at the moment operate at a fairly high level and require the user to try and map his data structures on to an existing schema. KeepSake provides low-level primitives which give the user much greater control over the disc and we believe that our approach gives the user much greater flexibility.
Our non-overwriting implementation gives a much simpler solution to the problem of maintaining data consistency than the conventional overwriting approach. We believe that our approach would be a useful starting point from which to consider implementing a distributed database.

References

[1] Peeling N.E., Morison J.D., Whiting E.V., ADAM: An Abstract Database Machine, RSRE Report No. 84007, 1984.

[2] Peeling N.E., Morison J.D., A Database Approach to Design Data Management and Programming Support for ELLA, A High-Level HDDL, CHDL 1985.

[3] Stanley M.S., An Evaluation of the Flex Programming Support Environment, RSRE Report No. 86003, 1986.

[4] Peeling N.E., Milner K.R., KeepSake: Algorithms and Implementation, RSRE Memo No. 4255, 1988.

[5] Rees S.J. A Persistent Heap for ALGOL68, Paper in preparation.

THIS PAGE IS LEFT BLANK INTENTIONALLY

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Report 88014		U/C

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
7784000	ROYAL SIGNALS & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN, WORCESTERSHIRE WR14 3PS

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

KEEPSAKE : A database kernel

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
Peeling N E	Milner K R		1988.12	22

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract

KeepSake is a set of disc management procedures that can be used to extend a programming language to provide data persistence. It provides efficient low-level read/write procedures and also allows flexibility in partitioning a database for garbage collection and multi-user read/write access. KeepSake does not impose a data schema on the user but can be used to support a number of database types (relational, hierarchical etc.).

THIS PAGE IS LEFT BLANK INTENTIONALLY